A Case Study into Predictable and Composable MPSoC Reconfiguration

Pranav Tendulkar
Verimag, University of Grenoble,
Centre Équation - 2, avenue de Vignate,
38610 Gières
France
Email: pranav.tendulkar@imag.fr

Sander Stuijk
Eindhoven University of Technology
Den Dolech 2,
5612 AZ Eindhoven
The Netherlands
Email: s.stuijk@tue.nl

***Abstract*—The number of applications running concurrently on a MPSoC is ever increasing. Moreover, the set of running applications is often unknown at design-time. Part of the resource allocation decisions must therefore be deferred to run-time. This requires a run-time manager to optimize the resource usage of the system to preserve energy and allow as many applications as possible to use the resources simultaneously. An effective resource manager should therefore be able to reconfigure the resource assignment of running applications. To this end, a run-time task migration mechanism is needed. A user should however not notice the reconfiguration, as this would impact the perceived quality of the system. Hence, the reconfiguration mechanism should provide timing guarantees on its operation and it should not interfere with other applications running on the same system (i.e., it should be predictable and composable). In this paper, we present a practical implementation of such a predictable and composable MPSoC reconfiguration mechanism. We demonstrate the use of this mechanism on a JPEG decoder whose tasks are migrated at run-time while running on a state-of-the-art MPSoC platform.**

***Index Terms*—Task migration, real time systems, timing guarantees**

I. Introduction

The number of applications which a user is running concurrently on a modern MPSoC is increasing rapidly. A user may, for example, use a mobile phone to watch a video that is being decoded using an MPEG-4 decoder while an MP3 decoder is used to decode the accompanying audio. At the same time, other background processes may be running on the same MPSoC to update the agenda or e-mail of the user. In many systems, the set of applications that may be active simultaneously is no longer known at design-time. Resources need to be assigned to applications at run-time. This task is handled by a run-time resource manager that is running on the MPSoC platform. Whenever an application is started or stopped, the resource manager should adapt the resource assignment of all running applications accordingly while considering multi-objective, potentially conflicting, optimization criteria (e.g., improve QoS and/or reduce energy consumption of the set of running applications). Many of these applications will have real-time constraints that need to be met even when the resource manager decides to reconfigure the resource assignment. To meet this requirement, both the decision process (i.e., which resource assignment to use) as well as the process to implement this decision (i.e., the actual resource reconfiguration) need to be done within a predictable and bounded amount of time. In other words, the run-time resource manager should implement a resource management and reconfiguration strategy that provides a predictable timing behavior. This paper presents such a predictable run-time resource manager on the CompSOC MPSoC platform [1].

At design-time, the design-flow presented in [2] can be used to map individual applications to the CompSOC platform. When possible and useful, the design-flow will compute for an application multiple possible configurations representing the available trade-offs between resource usage and quality. Configurations could for example provide a trade-off between the use of different resources, such as different processors, or trade-offs between processing and memory usage or through DVFS between energy consumption and resource utilization. The configurations determined at design-time are Pareto optimal trade-offs between the various quantities considered at run-time (i.e., resource usage, energy usage and quality). The configurations are determined at design-time on a per-application basis. The run-time resource manager must combine these configurations at run-time to investigate system-wide trade-offs. After combining the trade-off spaces of the individual applications, the resource manager can decide on the optimal system-wide configuration of all running applications and subsequently it can implement this decision by reconfiguring the individual running applications. As mentioned before, all of these steps need to be performed by the run-time resource manager while providing timing guarantees to the running applications. A resource manager which fulfills these constraints is introduced in this paper. The resource manager uses the predictable and composable MMKP heuristic presented in [3] to select an optimal configuration for all applications running on the platform. The configuration selected by this heuristic may trigger a reconfiguration of these applications. Our proposed run-time mechanism ensures that this reconfiguration process is completed within a bounded amount of time (i.e, the reconfiguration is predictable). Moreover, the timing behavior of applications of which the resource assignment is not changed will not be affected (i.e., the reconfiguration is composable). We demonstrate our run-time

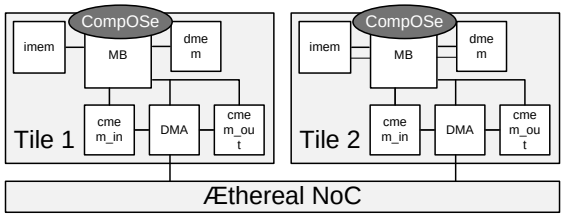


Fig. 1: CompSOC platform.

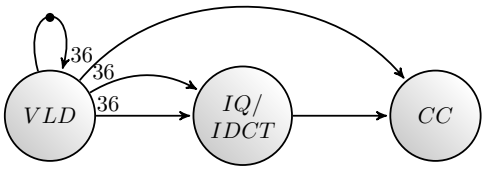


Fig. 2: SDFG of a JPEG decoder.

reconfiguration mechanism with a JPEG decoder whose tasks are migrated while the platform is active.

The remainder of this paper is organized as follows. Sec. II introduces the CompSOC MPSoC platform on which our run-time reconfiguration mechanism is implemented. The application programming model for this platform is described in Sec. III. The proposed reconfiguration mechanism is presented in Sec. IV. A case study in which our reconfiguration mechanism is used is described in Sec. V. Related work is discussed in Sec. VI and Sec. VII concludes this paper.

II. CompSOC platform

The CompSOC platform (see Fig. 1) has a tiled-based architecture in which a set of processing and memory tiles are connected to each other through the Æthereal network-on-chip [4]. Each processor tile contains a microblaze processor running the CompOSe real-time operating system [5]. CompOSe provides composable and predictable application scheduling. A processor tile contains also a non-shared local memory for instructions and data, as well as communication memories which are used by a DMA for communication with remote tiles. Memory tiles contain a memory sub-system that can be accessed from the processing tiles. In the scope of this work, memory tiles do not play an important role since they do not have to be considered in the reconfiguration process. Therefore, we will not discuss memory tiles in this paper.

The CompSOC platform provides a predictable and composable timing behavior to applications running on the platform [6]. In order to provide composability, it uses a composable scheduling strategy such as time-division multiplexing (TDM), where the presence or absence of requests from one application cannot affect when other applications are scheduled. In addition, it uses preemption after a fixed time to prevent one application from starving other applications. Furthermore, it delays scheduling requests till the end of a time slice to avoid that the early completion of one request will cause subsequent requests to be scheduled earlier. In order to provide predictability, it requires that all data needed for a request must be locally available and that there should be enough local storage space to store the response of this request. In combination with the use of predictable resources with bounded worst-case execution times and the use of a predictable TDM scheduler, it is possible to compute per task running on a resource a worst-case response time for this task. Complete details about this platform can be found in [1], [6].

III. Application Programming Model

Programming MPSoCs such as the CompSOC platform is a challenging task. To address this challenge, a model-based design approach has been developed to program the CompSOC platform [1]. This approach uses the dataflow Model-of-Computation (MoC) to describe the behavior of applications that must be implemented on the platform. Several variants of this dataflow MoC, which differ in their expressiveness, are supported by the design approach and platform. In this work, we assume that applications are implemented as a synchronous dataflow graph (SDFG) [7]. Fig. 2 shows an example SDFG. The nodes in this graph are called actors and they represent the functionality of the application. The actors communicate with each other by sending tokens (data-items) in FIFO order over the edges in a SDFG. An actor can be executed (fired) if sufficient tokens are available on all its input edges. An important property of a SDFG is that on each firing it consumes a fixed amount of tokens from all input edges and it produces a fixed amount of tokens on all its output edges. These amounts are called the rates and are annotated at the start and end of each edge (rates 1 are omitted for clarity). Some channels may contain initial tokens (indicated with a black dot). These initial tokens represent data-items in the application which have an initial value that is required for the computation performed by the actors.

IV. Run-Time Resource manager

As explained in Sec. I, the run-time resource manager consists of two main components (see also Fig. 3). The first component, the *system-level resource manager*, decides in which configuration each application must be running. Whenever a new application is started on the platform or a running application is stopped, this component may decide to reconfigure the running applications. The second component, the *application-level resource manager*, must implement this decision (i.e., it must migrate actors and edges between processors). As explained below, to build a composable run-time manager it is required that each application has its own application-level resource manager.

A. System-level resource manager

The system-level and application-level resource managers will be running on top of CompOSe. The system-level resource manager is implemented as a separate application running on the platform. Because of the composability offered by CompSOC, it is guaranteed that this part of the run-time

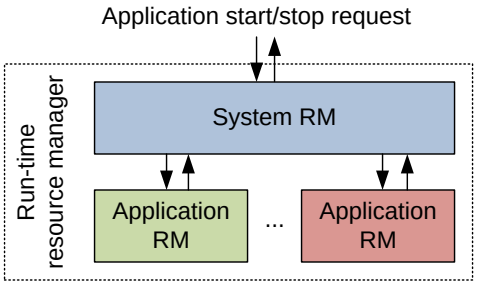


Fig. 3: Run-time resource manager (conceptual view).

reconfiguration mechanism will not interfere with the timing behavior of other applications.

The system-level resource manager can be implemented using an algorithm such as the MMKP heuristic presented in [3]. This algorithm decides on the optimal configuration of all applications in a bounded amount of time (i.e., it is predictable) while considering multiple optimization criteria. In this work, we use this MMKP heuristic in our system-level resource manager. An interested reader is referred to [3] for a detailed discussion on this algorithm. The main contribution of this work is the application-level resource manager, which implements the resource allocation decisions taken by the system-level resource manager within a bounded period of time.

B. Application-level resource manager

The implementation of a reconfiguration decision, which is handled by the application-level resource manager, may require that tasks from an application are migrated from one processing tile to another. This process is realized in our run-time reconfiguration mechanism as part of the application which is being reconfigured. From the perspective of the CompOSe operating system, our application-level resource manager is part of the application.

Our application-level resource manager uses a master-slave configuration (see Fig. 4). The master stores the active resource configuration of the application. Whenever the MMKP algorithm decides to reconfigure the application, the master will send a set of reconfiguration commands to the slaves. For this purpose, a pair of dedicated FIFOs is used between the master processor and each slave in the platform.

Since the whole reconfiguration process takes place during the TDM time slices allocated to the application, it is guaranteed that other applications will not be affected by the reconfiguration (i.e., the run-time resource manager is composable). In order to arrive at a predictable run-time mechanism it is however also important that the reconfiguration process itself can be completed within a bounded amount of time. Otherwise, the application which is being reconfigured may miss its own timing deadlines.

C. Migration Point

To reduce the task migration overhead, a reconfiguration can only be performed at specific moments during the execution of the application. In the SDF MoC, actors have no state that needs to be preserved across firings. Any state (data) that needs to be stored between firings should be stored explicitly as a token on a self-edge of an actor. Allowing task migrations only when an actor is not executing (firing) ensures that no state (other than the initial token on the self-edge) needs to be transferred. This implies, no processor context needs to be migrated, which reduces the task migration overhead considerably.

Whenever an actor is migrated to a different processor, the edges connected to this actor must also be reconfigured. Tokens that are present in these edges must then be migrated from the old edges to the newly created edges. Throughout the execution of the SDFG, the number of tokens in the edges may vary. Hence, the amount of tokens that needs to be transferred may vary depending on the moment when an actor would be migrated. As a result, the time needed to migrate a task and its connected edges might depend strongly on the number of tokens in these edges. In order to provide timing guarantees, design-time analysis techniques must take the worst-case situation into account when analyzing whether a particular reconfiguration would be feasible given the timing constraints of the application. When task migrations are allowed to occur at any moment when an actor is inactive, this could lead to very pessimistic estimates on the number of tokens that needs to be transferred. As a result, design-time analysis techniques would most probably indicate that many reconfiguration options cannot be performed within the given timing constraints. To address this issue, we use a special property of SDFGs. Since the production and consumption rates of the actors on the edges are constant in a SDFG it holds that after each actor has been fired for a certain number of times (called an iteration of the SDFG [7]), the token distribution in a SDFG returns to its original state. At this moment, the number of tokens in all edges connected to an actor are equal to the number of initial tokens. Since this is a well defined amount, it can be easily taken into account in the timing analysis performed at design-time. Therefore, our application-level resource manager will only migrate a task and its connected edges when the actor and all actors connected to the edges of this actor have completed an iteration. Note that it is insufficient if only the actor which will be migrated has completed an iteration. The actors that communicate with this actor should also be halted at the start of an iteration. Otherwise, tokens may be added or removed from these edges and the number of tokens in these edges would then be unequal to the number of initial tokens.

D. Task and FIFO Migration on CompOSe

When migrating an actor, the schedule on the processor on which this actor was originally running as well as the schedule on the processor to which the actor is moved must be changed. In CompOSe, actors from the same SDFG are scheduled using a static-order schedule. To modify these schedules, CompOSe requires that the complete schedule of all actors that belong to the same SDFG are removed from a processor

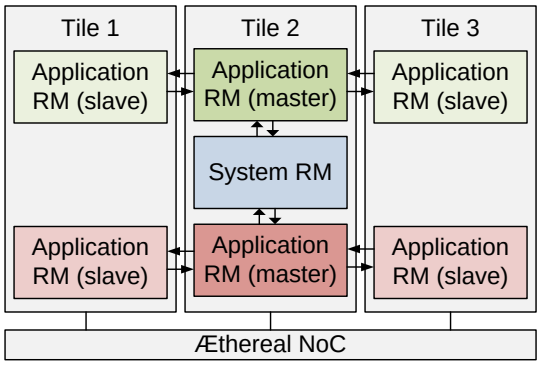


Fig. 4: Run-time resource manager (deployment view).

and subsequently a new schedule can be assigned to the actors. This implies that the application (SDFG) should be stopped on the processor from which the actor is migrated away as well as the processor to which the actor is moved. When migrating an actor, the edges (FIFOs) connected to the actor must also be migrated. This can be done through CompOSe by removing the old FIFOs and adding the FIFOs to the new migrated actor. If initial tokens are present in the old FIFO, they are retained and copied to the new FIFO. CompOSe requires that the application whose set of FIFOs is modified on a processor is not running during this reconfiguration. Hence, the application must be stopped on the processor which is left by the migrated actor, the processor receiving the migrated actor and all processors that run actors which have at least one FIFO edge to these two processors. From many practical situations, this will often imply that the application must be halted on all processors on which it is running. Considering this aspect and in order to reduce the number of messages exchanged between the master and slave components of our application-level resource manager, the resource manager will halt the application on all processors. Next, it will perform the reconfiguration. When this process is completed, the application will be resumed on all processors. Note that during this whole process, the other applications running on these processors are not interrupted which ensures that our run-time reconfiguration is composable.

The next section presents in detail the messages that are exchanged between the master and slave components in our application-level resource manager when reconfiguring an application. As explained in that section, each individual operation during the reconfiguration process (e.g., reconfiguration decision, actor migration, FIFO creation, etc.) can be performed in a bounded amount of time. Since the complete set of operations that needs to be performed in order to migrate an actor is known at design-time, it is possible to provide a timing guarantee on the completion of the complete reconfiguration. Hence, the run-time resource manager offers a predictable reconfiguration mechanism.

V. Case Study

A. JPEG Decoder Application

We demonstrate our resource manager with a case study in which some actors of a JPEG decoder are migrated from one processor to another. Fig. 2 shows the SDFG of our JPEG decoder. It consists of three actors, namely Variable Length Decoder (VLD), Inverse Quantizer combined with Inverse Discrete Cosine Transform (IQ/ICDT) and Color Conversion (CC). The VLD actor has state that needs to be preserved across firings. In between firings, this state is stored as an initial token on the self-edge connected to this actor. One pair of edges from the VLD to IQ/IDCT respectively CC actor is used to communicate JPEG header parameters (e.g., image size) between the actors. The remaining edge from the VLD to the IQ/IDCT actor and the edge from the IQ/IDCT to the CC actor are used to communicate the actual image data.

B. Run-Time Task and FIFO Migration

Fig. 5 illustrates the actor migration process. Initially the VLD and IQ/ICDT actors are running on the second tile and the CC actor is running on the first tile. Next to these actors, the second tile is also running the system-level resource manager as well as the master application-level resource manager for this application. The other two tiles are running a slave application-level resource manager for our JPEG decoder application. In this example, no other applications are active on the platform. At some point in time, the system-level resource manager decides to migrate the CC actor from tile 1 to tile 3. (This decision would normally be triggered when a new application is started on the platform, but for simplicity we assume in this example that the system-level resource manager takes this decision without any new application entering the system). Once the system-level resource manager has taken the decision to reconfigure the application, it informs the master application-level resource manager about this decision (step 1 in Fig. 5). Since this application-level resource manager is part of the JPEG application running on tile 2, it will be periodically scheduled on this tile. The first time it gets scheduled after the system-level resource manager took the decision to reconfigure, it will start the actual reconfiguration process. This process starts with sending a command to all tiles to halt the application (step 2 in Fig. 5). This is done by sending a message through the dedicated FIFOs between the master and slave application-level resource managers. Whenever an application-level resource manager is scheduled on a processor, it will check this FIFO to verify whether new commands are available in this FIFO. If so, these commands will be processed. Once the command to halt the application on the tile has been completed, the slave application-level resource manager will inform the master application-level resource manager by sending an acknowledgment (step 3 in Fig. 5). When all slaves has confirmed that the application has been halted and the application has also been halted on the tile running the master application-level resource manager, the resource manager continues with the next step of the

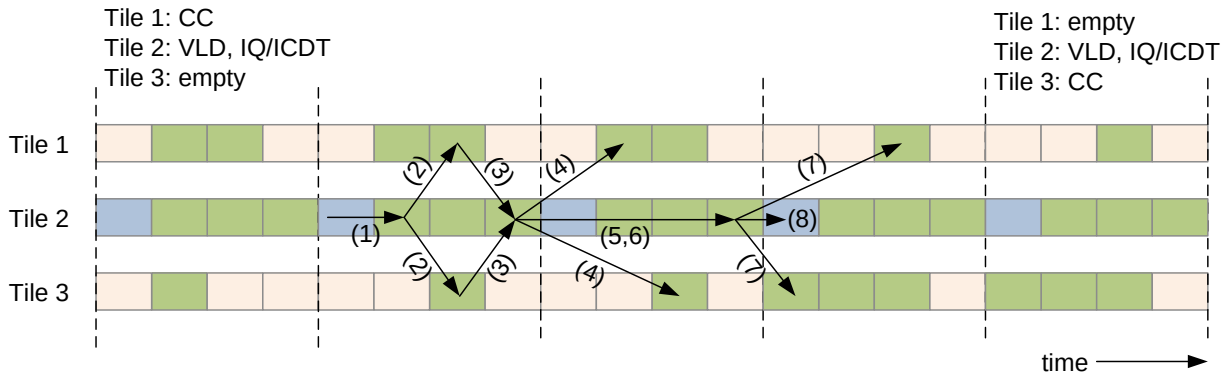


Fig. 5: Task migration (steps performed by resource manager).

TABLE I: Task migration overhead (in clock cycles).

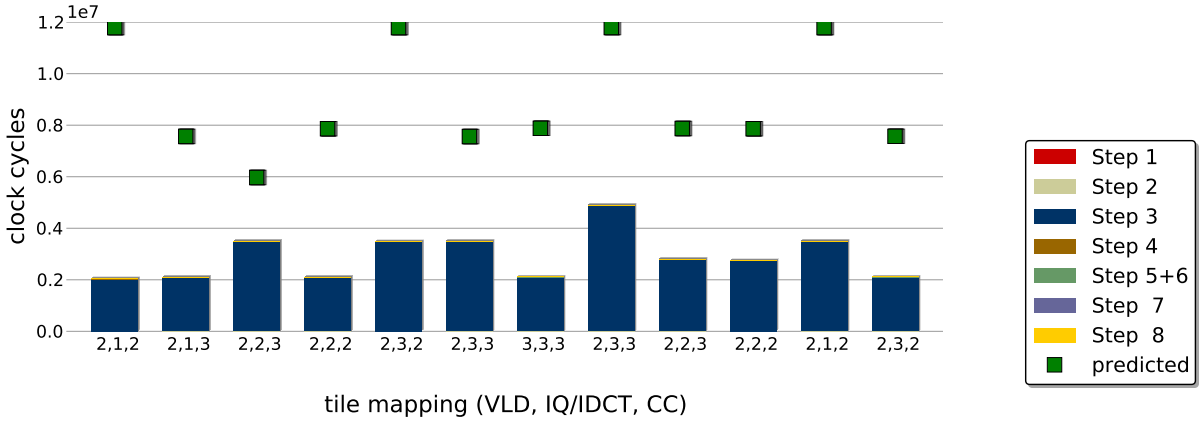
Step	Description	Master	Slave
1	Instruct application RM to reconfigure	50	n/a
2	Request removal of application from TDM	140	760
3	Remove application from TDM and ack.	App. dependent	
4	Resize TDM allocation	300	850
5/6	Add/remove FIFO	Tab. II	Tab. II
7	Add application to TDM	570	900
8	Inform system RM about completion	50	n/a

reconfiguration process. Since after reconfiguration there will be no actors of this application running on tile 1, the number of time slices allocated to the application can be reduced on tile 1 to just one (which is needed to periodically execute the slave resource manager). Furthermore, the number of TDM slices allocated on tile 3 may have to be increased. This is illustrated as step 4 in Fig. 5. In this step, the master resource manager instructs the slave resource managers to make this change in the TDM allocation. Step 5 involves the removal of the old FIFOs from tile 2 to tile 1 and in step 6 new FIFOs are created between tile 2 and 3. These FIFOs are immediately connected to the VLD actor on tile 2 and a new instance of the CC actor on tile 3. (Note that we assume that the instruction code of all actors is available on all tiles. Hence, no code migration is required). Next, the static-order schedule on tile 3 is updated (i.e., the CC actor is added to it). Subsequently (step 7), a message is sent to all tiles to resume the execution of the application. At this moment, the reconfiguration process of the application has ended and the master application-level resource manager confirms this to the system-level resource manager (step 8). This completes the complete reconfiguration process of the application.

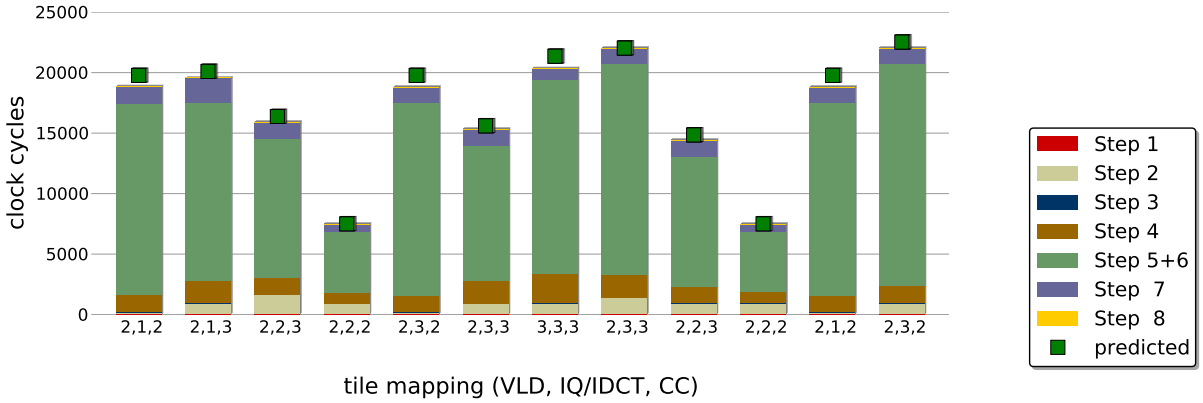
As mentioned before, in order to ensure a predictable reconfiguration process, it is important that each of the steps described above can be completed within a bounded amount of time. Our implementation ensures that this constraint is met. Tab. I lists the number of clock cycles needed to perform the various steps in the reconfiguration process. These times depend on the tile which ultimately needs to perform the operation. When the operation (e.g., removal of application from TDM) needs to be performed on a slave tile, then we need to consider the overhead of sending a message from the resource manager running on the master tile to the resource manager on a slave tile. The time required to remove an application from the TDM schedule (step 3) depends on the application. As explained before, an application may only be stopped on a tile when the application has completed a full iteration of the SDFG. In the worst-case, the application may have just started a new iteration on all tiles when a request to remove the application from the TDM schedule is received. In that case, a complete iteration of the SDFG must be finished before the request can be executed. Hence, the worst-case time needed to complete step 3 is bounded by the worst-case time needed to complete one iteration of the SDFG on the platform. Since all resources in the platform are predictable and since we assume that the worst-case execution time of all actors (tasks) are known, we can compute the worst-case time needed to complete step 3 at design-time. Tab. II shows the time required to reconfigure a FIFO. Depending on the tiles to which a FIFO is connected the time required to add or remove a FIFO varies. If the source and destination of a FIFO are both on the master tile, then there is minimal overhead to remove or add the FIFO. In this case, the overhead is limited to the allocating/freeing the data structure associated with the FIFO. When a FIFO is used to communicate between different tiles, for example between a tile running the master resource manager and a tile running a slave resource manager, the overhead will also include communication overhead to send a message to add or remove a FIFO and to send an acknowledgment after adding or removing the FIFO. The highest overhead occurs when a FIFO is connected between two different slave cores. In this scenario, the master application-level resource manager has to communicate messages between two different slaves and wait for their acknowledgments.

C. Results

It follows from Tab. I and Tab. II that the amount of time taken to complete a task migration depends on the old and new configuration of the application on the system. We



(a) Measurements with step 3 included



(b) Measurements without step 3 included

Fig. 6: Measured and predicted reconfiguration times.

TABLE II: FIFO add/remove overhead (in clock cycles).

Src	**Master**	**Master**	**Slave**	**Slave-1**	**Slave-1**
Destination	**Master**	**Slave**	**Master**	**Slave-2**	**Slave-1**
Time to remove FIFO	570	1235	1235	1735	975
Time to add FIFO	1280	4925	5050	8000	4400

have performed an experiment in which the JPEG decoder is reconfigured several times. Initially all three actors are mapped to tile 2 i.e., configuration 2,2,2 in Fig. 6 which indicates that the three tasks VLD, IQ/IDCT and CC are running on tile 2. As a first reconfiguration, the IQ/IDCT actor is migrated to tile 1. The left-most bar in Fig. 6 shows the run-time (in clock-cycles) needed to perform this reconfiguration. This run-time is measured using an actual implementation of our run-time reconfiguration mechanism on the CompSOC platform. The squared box above the bar indicates the worst-case reconfiguration time as computed using the run-times listed in Tab. I and Tab. II. The next bar in Fig. 6 shows the time needed to migrate the CC actor to tile 3 (configuration 2,1,2 to configuration 2,1,3). The other bars indicate other reconfiguration options that have been tested. Each time the label left to the bar indicates the configuration prior to the reconfiguration. The top part of Fig. 6 shows that the reconfiguration time is dominated by step 3. The bottom part of Fig. 6 shows the run-time of the reconfiguration process when excluding step 3. Comparing these two parts it is clear that the actual run-time of the reconfiguration process as well as the worst-case run-time of the reconfiguration process are dominated by step 3 (i.e., the worst-case of all other steps is always very close to the measured run-time for these steps). The fact that step 3 is the bottleneck in the reconfiguration process is not unexpected. As explained in Sec. IV, the run-time resource manager must in step 3 halt the execution of the application on all resources. In the worst-case this may require the application to execute a complete iteration before it can be halted. Even in the typical situation, it will require that many actors finish their execution before this step is completed. Hence, the long worst-case and measured run-times for this step. The only option to reduce the time taken by step 3 would be to relax the constraint that an application can only be reconfigured at the end of an iteration. However, as discussed in Sec. IV, this could in the worst-case lead to a large overhead in migrating tokens when reconfiguring FIFOs. An experiment with a small test application have confirmed that this overhead for most realistic applications far exceeds the worst-case time needed

to complete an iteration of the graph. From this we concluded that the choice to allow reconfigurations only at an iteration boundary is still the best option.

VI. Related Work

Task migration in a multi-processor system is performed for various reasons like thermal [8], [9], load-balancing [10], fault-tolerance [11] etc. We can classify the task migration work briefly in two types depending on the underlying architecture which is shared memory or distributed memory. We consider a scalable approach for multiprocessor with private memory for each processor, where the migration is done only at pre-decided checkpoints [12]. Taking the advantage of MoC, in our case the programmer need not explicitly store the context for migration.

[13] provides a comprehensive survey of the run-time resource managers. It divides the run-time resource manager into two parts; one which deals with decision making while the second is responsible for implementation of this decision. Casavant et. al. in [14] provides with the classification of different algorithms that can be used for resource manager. We use MMKP [3], which guarantees us predictability and flexibility. Owing to the private shared memory for each core, we implement the master-slave configuration of the resource manager, where the master is responsible for the decision making process, while the implementing the decision is performed collectively by all the responsible cores. This gives us an advantage of managing the data-structures simplistically [13]. We belong to the (type 3) adaptive applications classified in [13], where applications are allocated certain resources, as per the configuration of the application. The latter in turn is responsible to manage them efficiently.

[15] is also a master-slave configuration for task migration which has a copy of task in all the processor cores. They require sufficiently large queue size between the communicating actors to avoid the deadline misses in the application during the task migration. We migrate the application only the end of iteration, in a predictable one time slot in order to avoid any deadline misses and disturbances to other applications.

Almeida et. al in [10] propose tasks migration to provide work-load balance in multiprocessor system to optimize the throughput. There is no indication of amount of time required to migrate the task and its related resources. Further, their task migration is not capable of migrating stateful actors.

In [16], the author proposes locking of caches for hard-real time tasks which can provide predictable task migration. Their work focuses on cache based techniques, which become unscalable with increasing amount of processors on chip. Whereas we focus on private memory for each core, which has own instances of tasks running.

Hardware mechanisms [17]–[19] are proposed in order to offload the task of scheduling and migration to the hardware in order to save from the run-time overhead. However it is essentially a trade-off between the speed, flexibility and scalability of the system. Definitely, we do not outperform the hardware mechanisms, but our methodology gives a predictability and composability without requiring specialized hardware.

In AsyMOS [20], a different approach is taken, where the system management functions are handled by dedicated cores. Instead, we dedicate time-slots in our framework for management.

VII. Conclusion

This paper introduces a run-time reconfiguration mechanism that is able to provide timing guarantees on the time needed to migrate tasks and their communication channels in a MPSoC. The mechanism guarantees that the reconfiguration of one application will not affect the timing behavior of other applications running on the same resources. This enable predictable and composable MPSoC reconfiguration. The proposed run-time mechanism is demonstrated on a realistic MPSoC which is running a JPEG decoder.

References

[1] B. Akesson, S. Stuijk, A. Molnos, M. Koedam, R. Stefan, A. Nelson, A. Nedad, and K. Goossens, "Virtual platforms for mixed time-criticality applications: The CoMPSoC architecture and SDF3 design flow," in *Proceedings of workshop on Quo Vadis, Virtual Platforms? Challenges and Solutions for Today and Tomorrow*, 2012.

[2] S. Stuijk, M. Geilen, and T. Basten, "A predictable multiprocessor design flow for streaming applications with dynamic behaviour," in *Conf. on Digital System Design, DSD 10, Proc.* IEEE, 2010, pp. 548–555.

[3] H. Shojaei, A. Ghamarian, T. Basten, M. Geilen, S. Stuijk, and R. Hoes, "A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for cmp run-time management," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09. ACM, 2009, pp. 917–922.

[4] K. Goossens and A. Hansson, "The aethereal network on chip after ten years: goals, evolution, lessons, and future," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. ACM, 2010, pp. 306–311.

[5] A. Hansson, M. Ekerhult, A. Molnos, A. Milutinovic, A. Nelson, J. Ambrose, and K. Goossens, "Design and implementation of an operating system for composable processor sharing," *Microprocessors and Microsystems*, vol. 35, no. 2, pp. 246–260, 2011, special issue on Network-on-Chip Architectures and Design Methodologies.

[6] B. Akesson *et al.*, "Composability and predictability for independent application development, verification, and execution," in *Multiprocessor System-on-Chip — Hardware Design and Tool Integration*, M. Hübner and J. Becker, Eds. Springer, 2010, ch. 2.

[7] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[8] Y. Ge, P. Malani, and Q. Qiu, "Distributed task migration for thermal management in many-core systems," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. ACM, 2010, pp. 579–584.

[9] J. Jahn, M. Faruque, and J. Henkel, "CARAT: Context-aware runtime adaptive task migration for multi core architectures," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, march 2011, pp. 1–6.

[10] G. M. Almeida, S. Varyani, R. Busseuil, G. Sassatelli, P. Benoit, L. Torres, E. A. Carara, and F. G. Moraes, "Evaluating the impact of task migration in multi-processor systems-on-chip," in *Proceedings of the 23rd symposium on Integrated circuits and system design*, ser. SBCCI '10. ACM, 2010, pp. 73–78.

[11] P. K. Saraswat, P. Pop, and J. Madsen, "Task migration for fault-tolerance in mixed-criticality embedded systems," *SIGBED Rev.*, vol. 6, no. 3, pp. 6:1–6:5, Oct. 2009.

[12] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems-on-chip: a feasibility study," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, ser. DATE '06. EDAA, 2006, pp. 15–20.

[13] V. Nollet, D. Verkest, and H. Corporaal, "A safari through the MPSoC run-time management jungle," *J. Signal Process. Syst.*, vol. 60, no. 2, pp. 251–268, Aug. 2010.

[14] T. Casavant and J. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *Software Engineering, IEEE Transactions on*, vol. 14, no. 2, pp. 141 –154, feb 1988.

[15] A. Acquaviva, A. Alimonda, S. Carta, and M. Pittau, "Assessing task migration impact on embedded soft real-time streaming multimedia applications," *EURASIP Journal on Embedded Systems*, vol. 2008, no. 1, p. 518904, 2008.

[16] A. Sarkar, F. Mueller, and H. Ramaprasad, "Predictable task migration for locked caches in multi-core systems," in *Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, ser. LCTES '11. ACM, 2011, pp. 131–140.

[17] B. D. Theelen, A. C. Verschueren, V. V. R. Suárez, M. P. J. Stevens, and A. Nuñez, "A scalable single-chip multi-processor architecture with on-chip RTOS kernel," *J. Syst. Archit.*, vol. 49, no. 12-15, pp. 619–639, Dec. 2003.

[18] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu, "Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management," in *Proceedings of the international conference on Hardware/Software Codesign and System Synthesis: 2004*, ser. CODES+ISSS '04. IEEE Computer Society, 2004, pp. 48–53.

[19] T. Klevin, "Get realfast RTOS with xilinx FPGAs."

[20] S. Muir and J. Smith, "Asymos-an asymmetric multiprocessor operating system," in *Open Architectures and Network Programming, 1998 IEEE*, apr 1998, pp. 25 –34.